SELF OPTIMIZING FINITE STATE MACHINES FOR CONFIDENCE ESTIMATORS

Sandra J. Jackson and Martin Burtscher
Computer Systems Laboratory, Cornell University, Ithaca, NY 14853

ABSTRACT
Modern microprocessors are increasingly relying on speculation as a key technique for improving performance and reducing power, temperature and energy. Confidence estimators are necessary to prevent likely misspeculations. These confidence estimators are commonly realized using a Finite State Machine (FSM) called a saturating counter.

This paper presents a hardware method that allows FSMs to dynamically optimize their state transitions and confidence thresholds to improve CPU performance by automatically adapting to the current workload. The technique further allows the FSMs to continuously adjust to changing program conditions. These adaptable, self-optimizing confidence estimators are evaluated as a component in a value predictor on the C programs from the SPECcpu2000 benchmark suite. On average, the self-optimizing method achieves a miss rate reduction of 11% with a maximum of 47%. The self-optimizing confidence estimator can provide a speedup of 4%.

1. INTRODUCTION

Modern microprocessors rely more and more upon speculation techniques for improving performance and reducing power and energy. Speculation allows the CPU to forecast a piece of information it does not yet have, thus avoiding the delay imposed by waiting for the computation of the information. The CPU must eventually acquire the actual result, but it can continue execution with the speculated answer. If a prediction matches the computed outcome, the CPU saves valuable cycle time, power and energy because it can correctly use the results obtained during the speculation. However, should the CPU make an incorrect prediction the necessary recovery actions can be costly because instructions may need to be re-executed.

The purpose of a confidence estimator (CE) is to reduce the penalties incurred by incorrect predictions. As a program executes, the CE records a history of the success or failure of past speculations. If the number of times a prediction was correct is much greater than the number of times it was wrong, then the CE has a high confidence that the next prediction can be used without causing a need for recovery. If the confidence is low, the CPU discards the prediction and simply waits for the actual information to become available. Hence, confidence estimation allows the CPU to avoid the cost of using incorrectly predicted information. This improves the performance benefit of speculative techniques and saves power that would have been wasted running with the wrong information.

Currently, most CEs are realized using Finite State Machines (FSMs). The most commonly used FSM is the saturating up-down counter. It counts up for every prediction that is correct and decreases its value when a speculation is incorrect. It is saturating because once the counter has reached the highest value, the counter does not wrap around to zero when incremented but instead remains at the high value. A similar strategy applies to the lowest value. The higher the counter value the more confident the CE is that the next prediction will be correct.

The saturating up-down counter is commonly used because it provides reasonable performance across most programs. However, there are thousands of possible FSMs, and performance could be improved if an FSM tailored to the particular workload were used. Unfortunately, this would generally require an exhaustive search of all possible configurations. Even worse, in today's general-purpose processors one does not know ahead of time which programs will be run, making it infeasible to pick the best FSM in advance and statically implement it in hardware.

Genetic algorithms have been shown to find good solutions to problems using an evolutionary approach. A population of candidate solutions is used as a starting point, and the fitness of each individual is evaluated. Based upon these fitnesses a new generation of individuals is created by combining and changing the candidate solutions in the old generation through genetic operations. The goal is for each new population to be filled with individuals that are better than the ones in past generations.

In this paper we present a method inspired by genetic algorithms that allows a processor's hardware to automatically optimize the confidence estimators for the current workload. Our technique is transparent to software and employs a variable representation of an FSM that is implemented in hardware. Since the CE is constantly trying to improve itself during the execution of a program, it can also continuously adapt to changes in program behavior.

To evaluate our genetic method, we simulate a load value predictor [15] with a confidence estimator. We compare the performance of the genetic confidence estimator against the base saturating up-down counter, a saturating up-down counter with increased capacity, and a perceptron-based confidence estimator.

There are many types of load value predictors. Since there are numerous load instructions encountered during program execution, it is beneficial to apply predic-

tions to each load instruction separately. This is accomplished by including tables of load values whose entries are indexed using the PC and other information. CEs are employed, among other things, to differentiate between entries that yield accurate predictions and entries that do not.

We evaluate our adaptive, self-optimizing finite state machine CE using the C programs from the SPEC-cpu2000 benchmark suite. Our method substantially reduces the number of times the CE causes the processor to use a wrong value or not use a value that would have been correct. On average, the miss rate of the confidence estimator is reduced by 11%. The minimum miss rate reduction is 4% and the maximum is 47%. Cycle accurate studies show that our self-optimizing CE is capable of adding up to 4% of extra speedup.

2. RELATED WORK

As research in speculative methods increased, it became apparent that, while these methods could offer significant benefits, they also had the potential to slow down program execution if many of the speculations were incorrect. Grunwald et al. introduced confidence estimation [10] as a way to control the use of incorrect speculations and reduce the negative effects thereof. Calder et al. examine the use of a saturating counter confidence estimator and a history-based confidence estimator for value prediction [6]. It is important for the CE to be accurate; otherwise opportunities to use correct predictions can be missed or incorrect predictions will be used. Profiling can increase the accuracy of a CE by using a pattern history and identifying which patterns are followed by incorrect or correct subsequent predictions [4], [5]. However, the need to profile is usually undesirable.

Research for improving confidence estimators has also included using perceptrons as the CE component [3], [21]. While perceptrons improve prediction accuracy, cycle accurate studies show little or no speedup and actually exhibit a slowdown in some cases [21]. This is a result of the long time the perceptron takes to make a decision. The method we present in this paper requires fewer cycles than it would take to make a load value prediction, and therefore the processor knows whether to use a value immediately upon a prediction becoming available.

Methods for adaptable hardware often depend on the use of programmable hardware devices such as FPGAs; which can be used in combination with compiler and other techniques to adapt the existing hardware to the workload. Lau et al. study the possibility of specializing hardware to only include those resources that a program will need during execution [14]. Their approach involves setting up a library of possible architectures and having the compiler collect statistics for each model before deciding which one to use. They allow the architecture to be reprogrammed during execution as the application workload changes.

Another adaptable hardware approach involves creating physical features on a chip but allowing them to be enabled or disabled. The hardware detects when the current configuration experiences a decrease in performance and then tries several different configurations. The configuration that performs the best is then used until the performance decreases again [1], [2], [22]. Dhodapkar and Smith [7] developed a method to identify the current working set and select a configuration that matches the characteristics of that working set.

Fogel et al. first developed evolutionary programming [9]. Their approach represents possible problem solutions as FSMs. These FSMs were then evolved using random mutation of state transitions to find better machines. Holland [11] furthered the application of evolutionary techniques by creating Genetic Algorithms. This work provided a framework of successful genetic operations and multiple individual populations. These developments increased the success of genetic applications and spurred further research. For example, Emer and Gloy used genetic programming techniques to operate on a language describing predictor components and functions [8]. Applying genetic operations to the primitives in this language can automatically generate better predictors.

Sherwood and Calder designed FSM predictors tailored to specific workloads by profiling the desired program(s) and locating weaknesses in the existing predictor strategies [17]. Once a weakness has been identified, new FSMs may be desired for a whole program, a group of programs or just a few instructions. Using the profile information, a pattern definition is formed from which a language of regular expressions for the predictor history is developed. An FSM is then constructed from these regular expressions. The FSM can be translated by a tool and implemented in hardware. This works well for customized processors since the applications are known ahead of time. Sherwood and Calder also examine extending their approach to general-purpose processors [18].

Our approach is the first, pure hardware implementation of a genetically evolving problem solution that does not require intervention from the user or profiling. Confining the method to hardware adds additional challenges. It requires hardware sizes to be realistic and the design must be fast. These restrictions mean, for instance, that we need to deal with an extremely small population size compared to software genetic methods.

3. IMPLEMENTATION

A finite state machine confidence estimator contains two specifications, state transitions and the threshold at which the confidence estimator indicates that the prediction should be used. For example, Figure 1 shows a

2-bit FSM that could be used as a CE. It has a 3-bit input (consisting of the 2-bit current state and the 1-bit success or failure indication) and a 2-bit output that specifies the next state. The FSM transitions to a higher state when it sees a correct prediction. Once it reaches the highest state it remains there until it sees an incorrect prediction. Incorrect predictions cause transitions to lower states. The threshold specifies the state at and above which the confidence estimator allows the CPU to use the value prediction. If the threshold for the FSM is set at two then if the current state is equal to two or three the FSM would output a one, indicating that the predicted value should be used by the processor. The FSM in Figure 1 is commonly referred to as a saturating up-down counter.

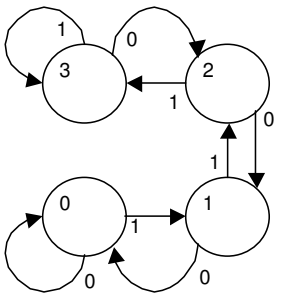

Figure 1: Saturating up-down counter finite state machine

correct?	state	next state
0	00	00
0	01	00
0	10	01
0	11	10
1	00	01
1	01	10
1	10	11
1	11	11

Figure 2: State transition table for the saturating up-down counter

Applying genetic operations in hardware requires the FSM to be represented as a set of bits. The next state column of the transition table in Figure 2 is the bit representation we use to specify the FSM. The last row of Figure 3 is simply the next state column transposed; this data can easily be stored in a register and changed to obtain new FSMs. The next state computation can be thought of as a lookup into an 8-entry, 2-bit memory. The bits in the columns "correct?" and "state" form the 3-bit memory address to access the 2-bit data, which is the next state value. If the bits stored in this memory are changed, so are the resulting state transitions. Figure 4 shows this operation in hardware, the two multiplexors simply function as an address decoder.

To find the best machines the threshold should be varied as well. In order to do this 01, 10, or 11, is prepended onto the bit string designating whether the CE will indicate to use the value prediction at a threshold of 1, 2, or 3. A threshold of 0, which would correspond to always using the predicted value, is not necessary because both always using a value and never using a value can already be expressed by state transitions that only lead to high or low confidence states. While different strings can essentially represent the same machines we choose this format to simplify the hardware and make it fast. Figure 5 shows the final FSM in bit string representation and highlights the hardware path that compares the threshold bits to the current state to determine the confidence.

A basic genetic method consists of the following steps [23]:

1. Create an initial population.
2. Calculate the fitness of each individual.
3. Apply genetic operations to obtain a new population.
4. Return to step 2 and repeat.

To genetically evolve better FSMs an initial population of FSMs must be created. The initial population is seeded with FSMs that provide good performance over a set of programs and that are sufficiently different from each other. Our initial population includes a saturating up-down counter and three additional FSMs. We found that a population size of four does not significantly limit the quality of the evolved FSMs. A small population size is essential to an efficient hardware implementation. For each new generation, the FSMs in the current population produce offspring to form the machines that will be evaluated in the new population.

A new generation of FSMs is genetically evolved after every interval of 262,144 (i.e., 2^{18}) executed loads. Since the size of the interval can affect performance we chose one that performed reasonably over all the benchmarks. Once the end of an interval is reached the current population is ranked based on the fitness of each individual. The fitness of each FSM is continuously incremented whenever that FSM makes a correct confidence estimation; this requires four counters plus an interval-length counter. The fitness function used is the accuracy of the current FSM. The higher the accuracy, the more fit the machine is for the current application. The best of the four FSMs replicates itself for the next generation and will be used as the confidence estimator in the next interval. The worst machine is re-

correct?	correct = 1				correct = 0			
state	11	10	01	00	11	10	01	00
next state	11	11	10	01	10	01	00	00

Figure 3: Illustration of how a state transition table can be represented as a string of bits.

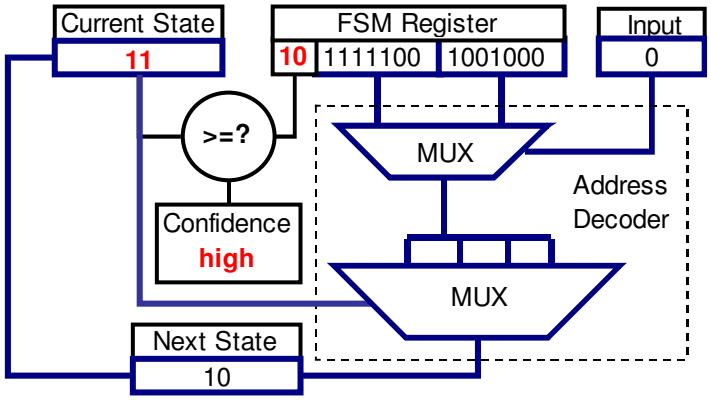

Figure 4: Hardware used to determine the next state is shown in bold.

moved from the population and cannot be picked as a parent for any of the three remaining FSMs to be generated.

Then two FSMs are selected as parents on which to apply genetic operations to create a new and hopefully better FSM. Selection is based on fitness, the higher the fitness of an individual the higher the probability that individual will be picked as a parent. Once two parents have been selected a crossover is performed, in which a random mask is generated from the least significant bits (LSBs) of the CPU's cycle counter. The bits that correspond to set bits in the mask are taken from one parent and the remaining bits from the other parent to form a new machine.

Next, a mutation is applied to the resulting bits. The bit to mutate is selected by loading an 18-bit circular shift register with a one in one bit position and zeros in the rest. The content of the register is rotated by one bit position every cycle. When a mutation is required this 18-bit value is xored with the FSM bits, which flips one bit.

This operation helps introduce variety into the new population if the machines become too similar. Applying the mutation operation immediately after the crossover also helps avoid exact copies of parents if the same machine gets selected as both parents. This process is repeated to create the third and fourth FSMs for the next generation. Since each individual FSM can be created in parallel this will only take a few cycles in hardware.

Unfortunately, the above implementation has a disadvantage. There are points during execution when the saturating up-down counter actually is the best machine, but it is lost after several generations in which it was not the best machine. To combat this effect, we limit the number of genetically evolving machines to three and replace the fourth machine with a static saturating up-down counter that gets evaluated along with the other FSMs. This modification requires the addition of a selector that chooses between using the saturating up-down counter or the current best genetic FSM as the confidence estimator. It is important to prevent the saturating up-down counter from making a copy of itself for the new generation; otherwise two copies of the same FSM would be evaluated. However, the saturating up-down counter needs to be included in the parent pool if it is currently the best machine.

The selector has the potential to further increase the accuracy if one selector is created per line of the CE. This is because the fitness of the FSMs is being evaluated globally, over all load instructions; therefore the resulting machine may not work well for a particular line. The selector per line allows the CE to take advantage of locality and use whichever machine would be best for the line being accessed. Figure 6 shows the storage required for the self-optimizing confidence

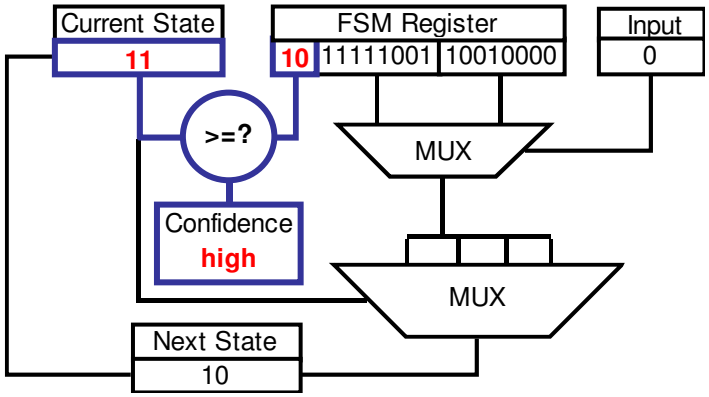

Figure 5: Hardware used to determine the confidence is shown in bold.

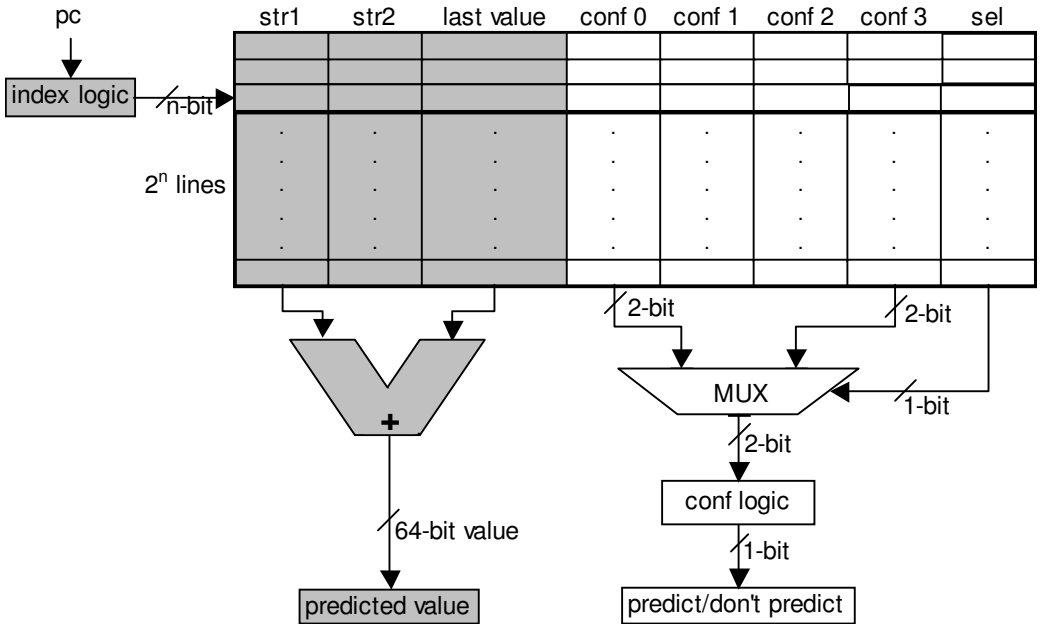

Figure 6: The hardware components of a stride 2-delta value predictor with a self optimizing confidence estimator.

estimator with the selector. The stride 2-delta value predictor [16] storage consists of the str1, str2 and last value fields. The table entries for conf 0, conf 1, conf 2, conf 3, and sel are each 2 bits wide. The saturating up-down counter state is stored in conf 3. Note that the conf logic block includes the logic seen in Figure 5 and an additional multiplexor to choose the appropriate FSM with which to make its decision.

Once the new generation of FSMs is ready, the hardware must prepare for the next interval. The fitness registers are reset to zero; there is one fitness register per FSM. The current state tables for each FSM are not reset because the tables are large and the new machines will need warm-up time. Not performing a reset also allows the best FSM, if it remained the same, and the saturating up-down counter to avoid a warm-up period.

Upon observing the machines generated over several benchmarks, we discovered that the FSM CEs would evolve away from one best machine, only to re-evolve the same machine again later in the program execution. This appears to be the result of program phases. Avoiding the need to re-evolve machines seen previously in the program is achieved by adding a small FSM victim cache that stores FSMs that have previously been the fittest machine. FSMs are retrieved from this cache when a drop in the fitness of the genetically generated population is detected.

The hardware chooses which victim to select from the cache by indexing the victim cache with a xor of the MSBs and LSBs of the saturating up-down counter fitness. The victim FSMs are also stored into the table in this manner. However, it is undesirable to retrieve a machine that would have done well in the past interval, which would occur if the saturating up-down counter's fitness from the presently completed interval were used as the storing index. This is because the FSMs are only stored into the cache after discovering they are no longer the best machine. Instead the victim FSMs are stored using the saturating up-down counter fitness from two intervals before the point at which they are stored into the victim cache. This way when the processor accesses the cache to find an FSM that might perform well in the future, it uses the current interval's saturating up-down counter fitness.

Table 1: MASE configuration

Processor	
Fetch/Issue/Commit Width	4/4/4
I-window/ROB/LSQ Size	64/128/64
Int/FP Registers	184
Execution Latencies	Alpha 21264
Branch Predictor	2-level 8K

Memory Subsystem	
Cache Sizes	64KB IL1, 64KB DL1, 1MBDL2
Cache Associativities	2-way L1, 4-way L2
Cache Latencies	2 cycles L1, 20 cycles L2
Cache Line Sizes	64B
Main Memory Latency	>= 400

Value Prediction Hardware	
Prediction Type	stride 2-delta
Table Size	2048
Value Prediction Latency	2 cycles

Genetic Method Support	
Table Size	2048
Line Size	10 bits
CE Latency	< 2cycles

This strategy retrieves an FSM from the victim cache and places it in the population that will be evaluated during the next interval. However, since only the saturating up-down counter and the best FSM can be used to make a prediction decision, the victim retrieved can only be chosen for decision-making starting in the interval following that in which it was re-inserted into the population.

4. METHODOLOGY

Cycle accurate simulations were performed with a modified version of MASE [13]. The baseline MASE architecture was supplemented with a stride 2-delta value predictor.

Table 1 shows the MASE configuration used in the cycle accurate studies. Upon encountering an incorrect value that was used the processor employs a re-execute strategy to repair the instructions that performed their computation with the incorrect value. A saturating up-down counter CE, a quadruple sized saturating up-down counter CE and a perceptron CE were implemented for comparison to our self-optimizing CE.

This study utilizes the SPECcpu2000 C benchmark programs. Experiments use a representative section of each benchmark determined by SimPoint [19]. Benchmarks are fast-forwarded to the beginning of the representative section and run until the end of the section using the reference inputs. The sections are 500 million instructions long.

5. RESULTS

In this section we present cycle accurate performance results and miss rate reductions comparing the self-optimizing confidence estimator to the baseline saturating up-down counter, a saturating up-down counter with four times the number of lines, and a perceptron confidence estimator.

5.1 Miss Rate Reduction

The miss rate reductions for the self-optimizing CE are compared with the best FSM for each program and the quadruple sized saturating up-down counter in Figure 7. "Missing" bars in the chart are zero and the baseline is the saturating up-down counter. Crafty, perlbmk, twolf and vortex receive the most benefit from the quadruple sized saturating up-down counter CE, but for most of these the self-optimizing CE is not far behind. In addition, there are cases in which the miss-rate reduction of the quadruple size saturating up-down counter is zero percent. Clearly, not all the miss predictions result from destructive aliasing, which would be lessened by increasing the table sizes.

It is also important to notice that in every program except gzip, both the self-optimizing CE and the self-optimizing CE with vcache outperform the best static FSM CE. Obviously, the ability to adapt to changing program behavior is beneficial. On average, the self-optimizing CE reduces the miss rate by 9.76%, adding the victim cache increases the average reduction to 11.86%. The minimum miss rate reduction is 3.97% and the maximum is 31.7%. The victim cache enhancement increases the minimum miss rate reduction to 6.53% and the maximum to 47.32%, beating the quadruple sized CE in 9 out of 15 programs and tying for one program.

The perceptron CE performs very well for most programs as is shown in Figure 7. The perceptron is able to learn patterns and beat the self-optimizing CE in all but 2 cases, crafty and twolf, in which the perceptron actually causes an increase in the miss rate over the saturating up-down counter. This is something that we have never observed with the self-optimizing CE. It is also important to note that the perceptron CE will take more cycles to make its prediction so the high miss rate reductions may not translate into speedup.

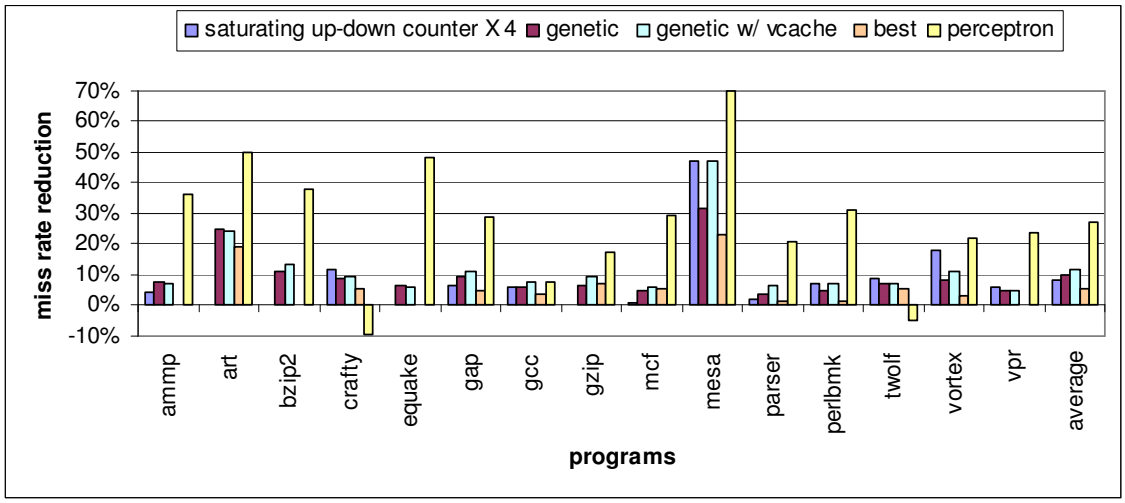

Figure 7: Miss rate reduction of the self-optimizing CE.

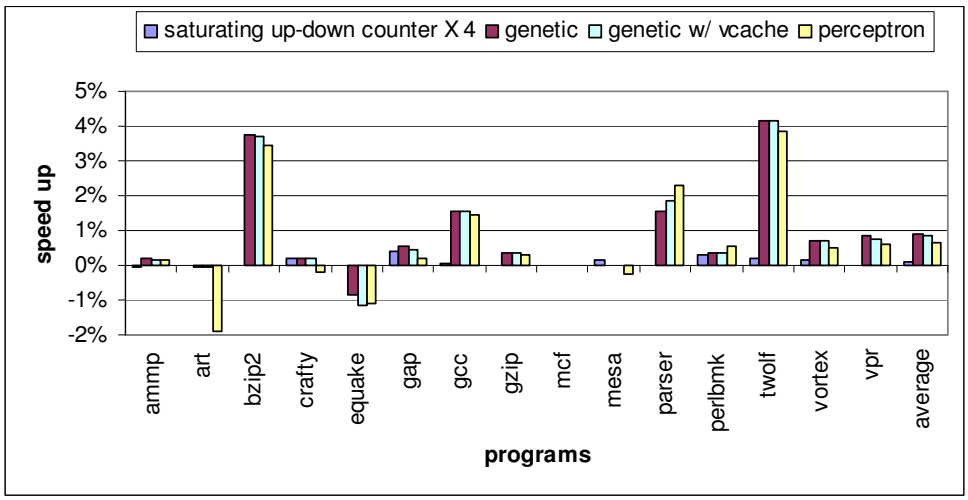

Figure 8: Speedup comparison of different CE methods.

5.2 Cycle Accurate Simulations

The self-optimizing confidence estimator is compared against the baseline saturating up-down counter, a saturating up-down counter with four times the number of lines, and a perceptron confidence estimator. To ensure a fair comparison and realistic sizing of the value predictor and CE, the stride 2-delta value predictor with a table size of 2048 was chosen as the value predictor to be evaluated with the CEs. This type and size of value predictor was determined to have realistic access times and hardware sizes by using a modified version of CACTI [20]. Cycle access times for each of the CEs were also computed using CACTI.

The perceptron confidence estimator keeps a global history of the value prediction correctness and applies the weights it stores to this history. A history size of 8 bits, as is used here, performs best with a weight size of six and a threshold of 29, as determined by Jimenez and Lin's formula [12]. The weight size and history size are used to determine the number of lines allocated to the perceptron predictor in order for it to be sized comparatively. The perceptron tables take 2 cycles to access plus an additional 3 cycles to compute the confidence.

The percent increase in the instructions per cycle (IPC), i.e., the speedup, is shown in Figure 8. The "missing" bars in mcf, mesa, parser and vpr are zero. The genetic method is capable of a 4% maximum speedup. While the perceptron CE shows a greater miss rate reduction than the self-optimizing CE, that performance does not translate into additional increase in IPC. This is most likely due to the perceptron's long latency to make a decision.

The saturating up-down counter X 4 gives minimal speed up, ammp shows a negligible slow down probably due to lines that were experiencing helpful aliasing. Since all the CEs are sized comparably, simply increasing the size of the CE is not a good solution, a better use of hardware real estate is made by the other methods.

The fitness function used in our self-optimizing CE rewards correct decisions without penalizing incorrect ones. As a result FSMs may evolve to use a larger portion of both correct and incorrect values counteracting any speed up seen from an increase in correct values. This may be why equake and art show slowdowns for the genetic methods.

6. FUTURE WORK

Some FSMs used in processor design are larger than 2 bits, applying the self-optimizing FSM method to those larger machines could result in more opportunity for adaptation. Obtaining reasonable changes in FSM performance might be more difficult for larger machines because each additional bit greatly increases the number of FSMs that can exist. Keeping that fact in mind, a small population size may not be able to evolve larger machines efficiently.

It might be possible to improve how the victim cache identifies which FSM to reintroduce into the population by taking advantage of techniques being developed in program phase research. This could be especially beneficial if the next phase could be predicted and an appropriate machine could be brought in and used immediately.

Additional possibilities include applying the self-optimizing FSM to other areas where FSMs are used, such as branch prediction, cache management, power control, etc. In areas like branch predictors where the FSMs are already extremely accurate, adding the storage necessary to implement a genetic method may not be practical.

7. CONCLUSIONS

We have shown that it is feasible to implement a ge-

netically adaptive method for evolving CEs fully in hardware. The need to keep the population size small to maintain manageable hardware sizes does not render a genetic method useless. Additionally, the hardware implementation allows adaptation to program phases, which leads to better miss rate reduction than a single static machine. In fact, our adaptive self-optimizing CE for value prediction can reduce the miss rate by an average of 11%, with a maximum reduction of 47% and is capable of delivering a 4% speedup.

8. ACKNOWLEDGMENT

This work was supported in part by a gift from Intel Corporation.

9. REFERENCES

[1] D. Albonesi. Dynamic IPC/Clock Rate Optimization, *Proc. of the 25th Int'l Symposium on Computer Architecture*, July 1998.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Architectures. *Proc. of the 33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[3] M. Black and M. Franklin. Perceptron-based Confidence Estimation for Value Prediction. *Proc. of the Int'l Conf. on Intelligent Sensing and Information Processing*, 2004.

[4] M. Burtscher and B.G. Zorn. Load Value Prediction Using Prediction Outcome Histories. *Univ. of Colorado at Boulder, Computer Science, Technical Report CU-CS-873-98*. Nov. 1998.

[5] M. Burtscher and B.G. Zorn. Prediction Outcome History Based Confidence Estimation for Load Value Prediction. *Journal of Instruction-Level Parallelism*, Vol. 1. May 1999.

[6] B. Calder, G. Reinman, D. M. Tullsen. Selective Value Prediction, *Proc. of the 26th Annual Int'l Symposium on Computer Architecture*, pp. 64-74, May 01-04, 1999, Atlanta, Georgia.

[7] A.S. Dhodapkar and J.E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. *ACM SIGARCH Computer Architecture News*, Volume 30 Issue 2, May 2002.

[8] J. Emer and N. Gloy. A Language for Describing Predictors and its Application to Automatic Synthesis. *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*, June 1997.

[9] L J. Fogel, A.J. Owens and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.

[10] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence Estimation for Speculation Control. *25th Int'l Symposium on Computer Architecture*. June 1998.

[11] J.H. Holland. *Adaption in Natural and Artificial Systems*, University of Michigan Press, 1975.

[12] D.A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. *Seventh Int'l Symposium on High Performance Computer Architecture (HPCA-7)*, Jan. 2001.

[13] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. *IEEE Int'l Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[14] D. Lau, M. Luttrell, I. Pines, and W.H. Mangione-Smith. A Framework for Implementing Customized VLIW Architectures on Programmable Hardware. *Proc. of the 1999 Workshop on Reconfigurable Computing, WoRC'99*, Oct. 1999.

[15] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value Locality and Load Value Prediction. *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[16] A. Mendelson and F. Gabbay. Speculative Execution Based on Value Prediction. *Technical Report, Technion* 1997.

[17] T. Sherwood and B. Calder. Automated Design of Finite State Machine Predictors. *UCSD Technical Report, CS2000-0656,* June 2000.

[18] T. Sherwood and B. Calder. Automated Design of Finite State Machine Predictors for Customized Processors. *28th Annual Intl. Symposium on Computer Architecture*, June 2001.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[20] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power and Area model. *Technical Report*, 2001.

[21] A. Thomas and D. Kaeli. Value Prediction with Perceptrons. *VPW2: Second Value-Prediction and Value-Based Optimization Workshop, Affiliated with ASPLOS XI*, Oct. 2004.

[22] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau and X. Ji. Adapting Cache Line Size to Application Behavior. *Int'l Conf. on Supercomputing*, July 1999.

[23] D. Whitley. A Genetic Algorithm Tutorial. *Colorado State Univ., Dept. of Computer Science, Technical Report CS-93-103*. Nov. 10, 1993.